Classification d'Images Sportives par Deep Learning

Gaétan Dumas, Morgan Jowitt
Université Paris 1 Panthéon Sorbonne — Master MOSEF

Avril 2025

---

1 Introduction

Ce projet vise à concevoir un système de classification automatique d'images sportives à l'aide de modèles de vision par ordinateur. Plusieurs architectures profondes pré-entraînées, dont **ResNet18**, **VGG16** et **EfficientNet-B0**, ont été explorées et fine-tunées sur un dataset open-source issu de la plateforme Hugging Face, regroupant plus de 50 catégories d'activités sportives. L'objectif final est de proposer une application simple et interactive permettant d'identifier automatiquement le sport représenté sur une image.

2 Présentation des données

Le jeu de données contient des images réparties en 100 classes sportives, de manière relativement équilibrée. La figure 1 montre la distribution des classes dans l'ensemble d'entraînement.

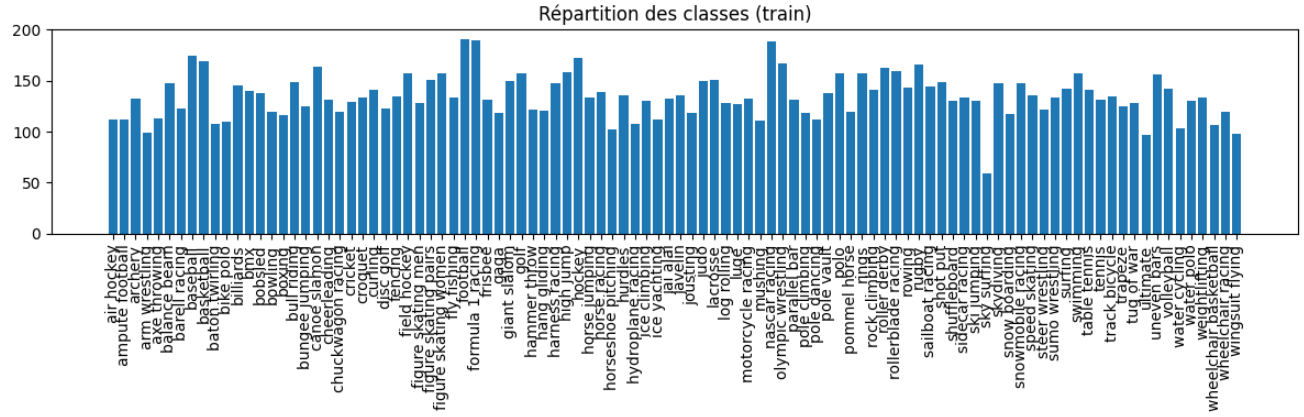

FIGURE 1 – Répartition des classes dans le jeu d'entraînement

Chaque image est accompagnée d'un label indiquant le sport pratiqué. Nous avons au total **13 492 images** dans l'ensemble d'entraînement.

Les classes sont globalement bien réparties, mais on note quelques déséquilibres : certaines catégories très visuelles comme **football** (191 images), **formula 1 racing** (190 images) ou **nascar racing** (189 images) sont particulièrement représentées. À l'inverse, des classes plus rares comme **sky surfing** (59 images), **ultimate** (97 images) ou **wingsuit flying** (98 images) comptent moins d'occurrences.

3 Prétraitement et chargement des données

Avant l'entraînement, chaque image est soumise à une série de transformations standardisées qui facilitent l'apprentissage et assurent la compatibilité avec les modèles pré-entraînés sur ImageNet. Ces transformations sont réalisées à l'aide du module `torchvision.transforms`, et consistent en trois étapes principales :

- **Redimensionnement** : les images sont converties à une taille fixe de `224x224` pixels via `transforms.Resize((224, 224))`, ce qui garantit l'uniformité des dimensions à l'entrée du réseau (essentiel pour les modèles convolutifs).
- **Conversion en tenseurs** : les images PIL sont converties en tenseurs PyTorch avec `transforms.ToTensor()`, transformant les pixels de l'intervalle `[0, 255]` vers `[0.0, 1.0]`.
- **Normalisation** : chaque canal (R, G, B) est centré et réduit avec la moyenne et l'écart-type de 0.5 :
  `transforms.Normalize([0.5, 0.5, 0.5], [0.5, 0.5, 0.5])`

Cela permet d'aligner la distribution des pixels avec celle des modèles pré-entraînés sur ImageNet, tout en accélérant la convergence.

Ces transformations sont appliquées automatiquement lors du chargement des images à l'aide de `torchvision.datasets` `ImageFolder`, qui classe automatiquement les images selon la structure du répertoire (une classe par dossier). Voici un extrait du code utilisé :

```
transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize([0.5]*3, [0.5]*3)
])

train_data = datasets.ImageFolder(".../train", transform=transform)
train_loader = DataLoader(train_data, batch_size=32, shuffle=True)
```

Dataloader et batching Nous utilisons `DataLoader` pour gérer efficacement les données :

- **Batching** : les images sont regroupées en lots de 32 pour paralléliser l'entraînement.
- **Shuffling** : les données d'entraînement sont mélangées à chaque époque pour éviter le sur-apprentissage sur l'ordre.
- **Chargement GPU** : les tenseurs sont envoyés directement sur le GPU pour maximiser les performances.

L'utilisation combinée de ces éléments permet de constituer un pipeline de données efficace, cohérent et prêt à l'entraînement sur les architectures de deep learning modernes.

4 Modèles testés et performances

Nous avons comparé trois architectures classiques de vision par ordinateur : **EfficientNet-B0**, **VGG16** et **ResNet18**. Tous les modèles sont initialisés avec des poids pré-entraînés sur ImageNet, puis fine-tunés sur notre dataset pendant 5 époques. Les métriques sont calculées sur l'ensemble de test.

- **EfficientNet-B0** obtient la meilleure performance globale, avec une précision de **98.00%** et un F1-score de **0.9797**. Il utilise seulement 4,1M de paramètres mais nécessite un temps d'entraînement plus long (**1h38min**).
- **ResNet18**, utilisé comme base du projet, atteint une précision de **97.20%** et un F1-score de **0.9712**, pour un temps d'entraînement très rapide (**7min30**) et un compromis intéressant entre performance et légèreté (11,2M de paramètres).
- **VGG16** propose une précision légèrement inférieure (**93.80%**, F1-score : **0.9363**), malgré un nombre de paramètres bien plus élevé (134,6M). Son temps d'entraînement est d'environ **21 minutes**.

TABLE 1 – Comparatif des modèles entraînés

Modèle	Accuracy	F1-score	Temps (s)	Nb. paramètres
EfficientNet-B0	98.00%	0.9797	5897.92	4.1M
ResNet18	97.20%	0.9712	449.25	11.2M
VGG16	93.80%	0.9363	1276.99	134.7M

En résumé, ResNet18 offre le meilleur compromis pour une application temps réel ou embarquée, tandis qu'EfficientNet-B0 reste le plus performant en termes de précision si le temps de traitement n'est pas une contrainte majeure.

Surveillance de l'overfitting

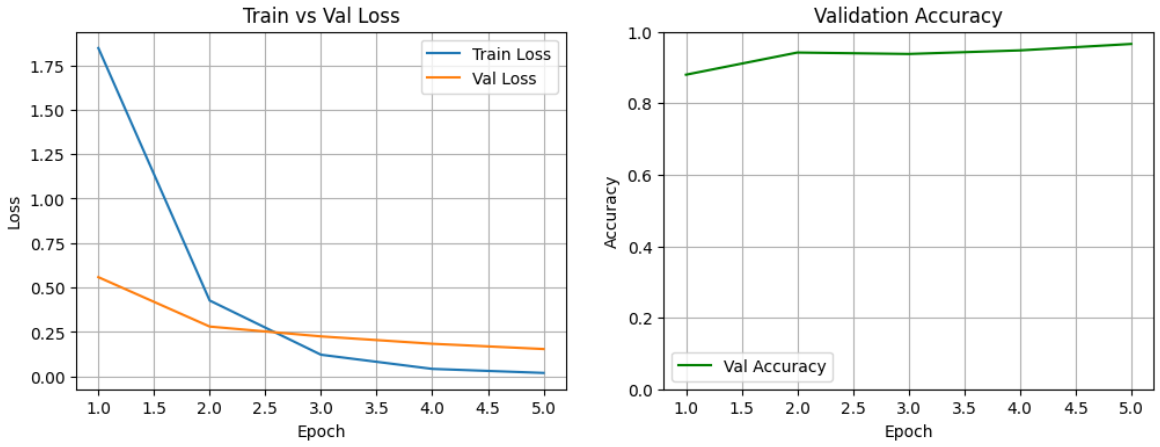

FIGURE 2 – Courbes de perte (entraînement vs validation) et précision en validation

La figure 2 illustre l'évolution des pertes d'entraînement et de validation ainsi que de la précision sur l'ensemble de validation au fil des époques.

On observe une diminution rapide de la perte d'entraînement, passant de près de 1.8 à moins de 0.05 en seulement cinq époques. En parallèle, la perte de validation diminue également mais de manière moins marquée, pour se stabiliser autour de 0.15 à partir de la 3^{e} époque. La précision de validation reste quant à elle élevée (au-dessus de 95

Ce comportement suggère un début de sur-apprentissage : le modèle continue à s'améliorer sur les données d'entraînement sans gains significatifs sur les données de validation. Pour limiter cet effet, plusieurs stratégies pourraient être envisagées :

- **Early stopping** : arrêter l'entraînement dès que la perte de validation cesse de diminuer ;
- **Régularisation** : via du dropout ou une pondération L2 (weight decay) ;
- **Augmentation de données** : pour enrichir la diversité du jeu d'entraînement.

Dans notre cas, l'overfitting reste limité, mais ces indicateurs encouragent à la prudence si le modèle devait être déployé sur des données plus variées ou bruitées.

Ressources matérielles et temps d'entraînement

Les temps d'entraînement indiqués ci-dessus ont été obtenus à l'aide des GPU proposés gratuitement sur Google Colab (principalement des NVIDIA T4 ou K80). L'utilisation d'un GPU a un impact majeur sur la rapidité d'entraînement : les opérations matricielles nécessaires aux réseaux de neurones profonds sont massivement parallélisées, ce qui permet de diviser par 10 à 100 le temps de calcul par rapport à un CPU classique.

À titre indicatif, un modèle comme ResNet18 s'entraîne en environ 7 minutes sur GPU, contre plusieurs dizaines de minutes voire plus d'une heure sur CPU. Pour des architectures plus lourdes comme VGG16 ou EfficientNet-B0, l'entraînement sur CPU pourrait nécessiter plusieurs heures, rendant difficile l'expérimentation rapide.

Ce projet a donc été mené entièrement sur GPU afin de garantir une itération efficace et exploiter les capacités optimisées de calcul des bibliothèques PyTorch.

5 Interprétabilité du modèle

Afin de mieux comprendre les décisions du modèle, nous avons utilisé la méthode **Grad-CAM** (Gradient-weighted Class Activation Mapping) qui permet de visualiser les régions de l'image ayant le plus influencé la prédiction finale.

Cette approche consiste à extraire les gradients des dernières couches convolutives du réseau pour générer une carte d'activation sur l'image d'entrée. Ces cartes permettent de valider l'attention du modèle sur les éléments pertinents (ballons, joueurs, lignes de terrain).

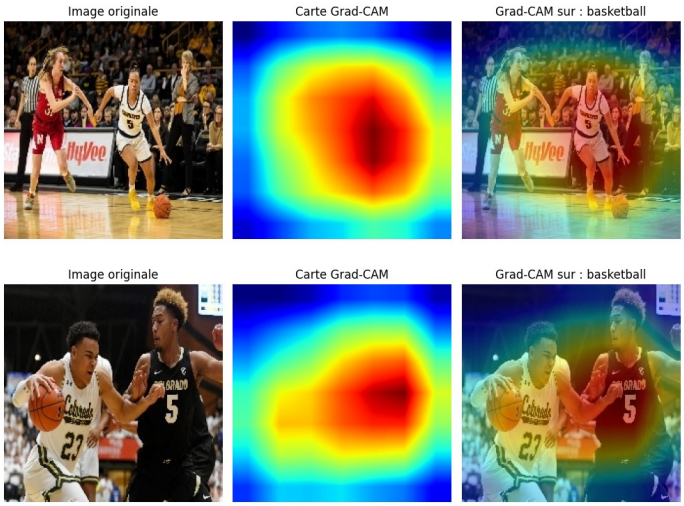

FIGURE 3 – Exemple de Grad-CAM : image originale (gauche), carte d'activation (centre), superposition (droite)

On observe que dans le cas du basket (figure 3), le modèle concentre son attention sur le ballon et les joueurs en mouvement, validant l'interprétation correcte du contexte sportif.

6 Analyse des erreurs

Malgré les performances globales élevées du modèle, certaines classes sportives restent difficiles à distinguer. Pour mieux comprendre ces limites, nous avons extrait les **10 classes les moins bien prédites** en analysant la diagonale de la matrice de confusion normalisée. Le résultat est présenté en figure 4.

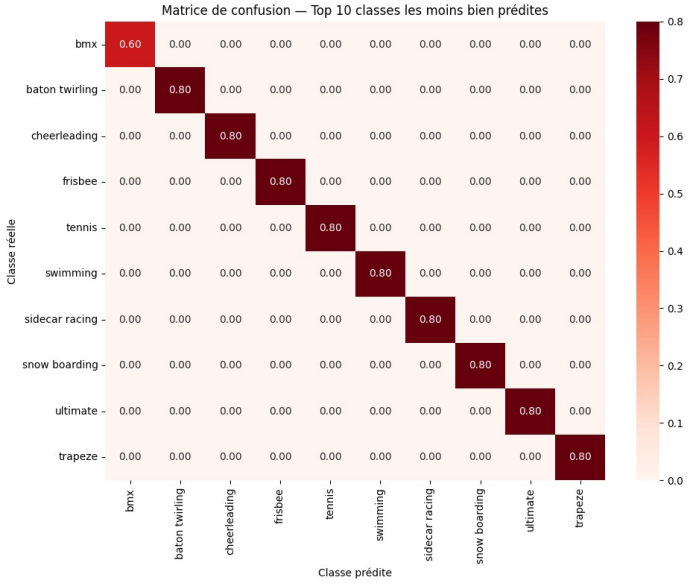

FIGURE 4 – Matrice de confusion — Top 10 classes les moins bien prédites

Parmi les erreurs fréquentes :

— **bmx** est confondu avec des sports aériens comme *skydiving* ou *bike polo*,
— **cheerleading** est pris pour du *basketball* (dû au contexte du terrain),
— **swimming** est assimilé à du *water polo* (éléments aquatiques communs),
— **frisbee** est mal identifié comme *figure skating women*, une confusion atypique probablement liée aux postures ou arrière-plans colorés,
— **baton twirling**, **trapeze** ou **snow boarding** sont parfois mal classés à cause d'une similarité dans les tenues ou les postures athlétiques.

Ces erreurs s'expliquent souvent par des contextes visuels ambigus, des arrière-plans chargés ou des classes visuellement proches. La figure 5 illustre ces limites à travers des images mal classées avec leurs labels réels et prédictions.

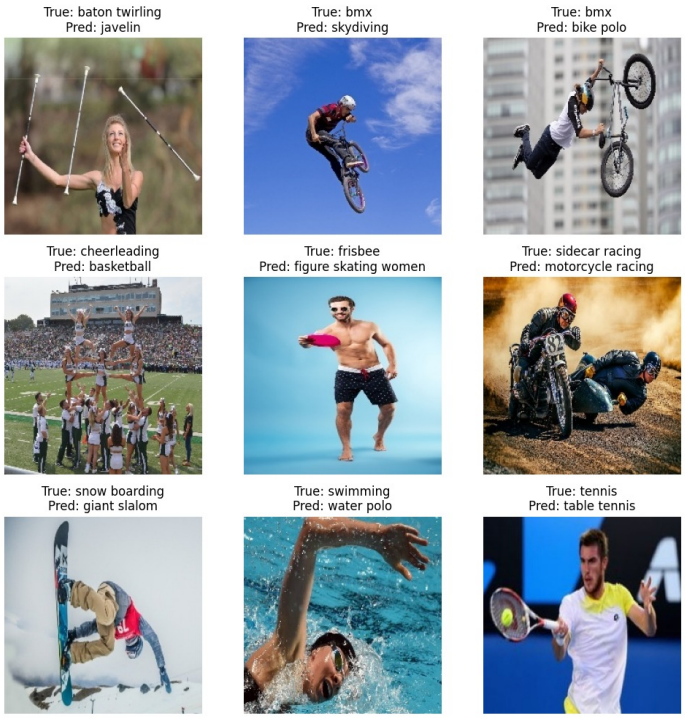

FIGURE 5 – Exemples visuels d'erreurs : label réel (haut), prédiction du modèle (bas)

Cette analyse met en évidence les limites perceptives du modèle face à des sports similaires visuellement. Elle souligne l'importance d'approches complémentaires telles que l'interprétabilité (cf. section Grad-CAM) ou la data augmentation pour renforcer la robustesse sur ces classes rares ou ambigües.

7 Application web

Dans une optique de mise en production et d'interactivité, nous avons développé une application web en Python à l'aide de la bibliothèque `Streamlit`. Cette interface légère permet à tout utilisateur de soumettre une image au modèle et d'obtenir en retour une prédiction parmi les 100 classes sportives disponibles.

L'application applique automatiquement les mêmes étapes de pré-traitement que lors de l'entraînement (redimensionnement à `224x224`, normalisation avec les moyennes d'ImageNet), garantissant ainsi la cohérence entre les phases d'apprentissage et d'inférence.

Une fois l'image transformée, le modèle retourne les probabilités pour chaque classe via une fonction `softmax`, et les cinq classes les plus probables sont affichées à l'utilisateur, accompagnées de leurs scores de confiance. Une visualisation Grad-CAM est également intégrée pour renforcer l'interprétabilité des prédictions.

8 Conclusion

Ce projet a permis de développer un système performant de classification d'images sportives, reposant sur plusieurs architectures fine-tunée sur un dataset riche de 100 classes. Malgré la complexité visuelle des données, le modèle atteint une précision supérieure à **97%**, avec des temps d'entraînement très raisonnables pour le **ResNet18**.

Grâce à une application interactive développée avec `Streamlit`, l'utilisateur peut facilement obtenir une prédiction ainsi qu'une visualisation des zones d'attention via **Grad-CAM**, renforçant l'interprétabilité du système.

Les résultats encourageants ouvrent la voie à plusieurs améliorations, notamment via l'augmentation de données, l'essai de modèles plus avancés (ViT, ConvNeXt) ou le déploiement sur mobile ou API web.